UML2Alloy Reference Manual

UML2Alloy Version: 0.52

Kyriakos Anastasakis

The University of Birmingham
School of Computer Science

Please consider the environment before deciding to print this document!

Contents

List Of Figures iii

1 Introduction **1**

1.1 Minimum Requirements 1

2 UML Profile for Alloy **2**

2.1 Analysis 2

2.2 ScopedElement 2

2.3 Singleton 3

2.4 Enforce 3

2.5 Simulation 3

2.6 Assertion 3

3 Transforming UML **4**

3.1 Packages 4

3.2 Classes, Subclasses and Types 4

3.3 Enumerations 5

3.4 Associations 5

3.5 Class Attributes 5

3.6 Class Methods 6

4 Transforming OCL **7**

4.1 Invariants 7

4.2 Operations 7

4.3 OperationCallExps 8

4.3.1 And/Or 8

4.3.2 Dot 8

4.4 If 8

4.4.1 Implies 8

4.5 Iterator Expressions 8

4.6 notEmpty/isEmpty 9

4.7 size 9

4.8 union 9

4.9 Including/Excluding 9

4.10 Select . . . 9
4.11 Includes/IncludesAll . . . 9
4.12 Excludes/ExcludesAll . . . 9
4.13 Not . . . 9
4.14 Variable Expressions . . . 9
4.15 Integer Expressions . . . 10
4.16 oclIsKindOf . . . 10
4.17 Boolean Literal . . . 10

5 UML2Alloy: Usage Manual 11
5.1 Installation . . . 11
5.2 Using UML2Alloy . . . 11
5.2.1 Scopes Tab . . . 11
5.2.2 Commands Tab . . . 12
5.2.3 Transform and Execute . . . 12

6 Examples 15
6.1 Two Unidirectional vs One Bidirectional Associations . . . 15
6.2 Precise UML for solving Sudoku . . . 17
6.3 Man-in-the-middle . . . 18
6.3.1 Unprotected . . . 19
6.3.2 TLS Protected . . . 19

7 Developer API 21
7.1 Defining a UML Model . . . 21
7.2 Transforming . . . 24

A XMI Compatibility 26

B Complete UML Model of Sudoku Puzzle 27
B.1 Generated Alloy Model . . . 28

References 33

List of Figures

2.1 UML Profile for Alloy Stereotypes 2

5.1 Files Tab of UML2Alloy 11
5.2 Scopes Tab of UML2Alloy 12
5.3 Commands Tab of UML2Alloy 12
5.4 Transform Tab of UML2Alloy 13
5.5 Selecting the Command to Execute 13

6.1 Example UML model of two classes connected with two unidirectional and one l
6.2 Exporting model to XMI using ArgoUML 16
6.3 Selecting that the areSame and bidirectional statements are Assertions 16
6.4 Available Commands to Execute 17
6.5 Counterexample 17
6.6 A Sample Sudoku Puzzle 18
6.7 A Class Diagram representing the Sudoku puzzle 18
6.8 Solution of the sample Puzzle of Figure 6.6 19
6.9 Man in the Middle Class Diagram (Unprotected) 19
6.10 Man in the Middle Class Diagram (TLS) 20

7.1 Package ‘org.uml2alloy.umlmmsimplifiedprofile’ 22
7.2 Example UML Model 23

1 Introduction

This document is the user manual of UML2Alloy version 0.52 [2]. The transformation rules are based on Alloy version 4 and UML version 2.0.

UML2Alloy is the outcome of research which attempts to formalise UML using Alloy [5]. UML2Alloy translates UML Class Diagrams enriched with OCL constraints to Alloy. Alloy models can be automatically analysed using the Alloy Analyzer [1]. The current version of UML2Alloy performs the transformation to the Alloy language and creates a text file with the Alloy model. Since version 0.5 experimental support for executing the generated Alloy model, from within UML2Alloy has been added. This is possible by utilising the Alloy Analyzer API. Currently, this feature is compatible with Alloy Analyzer 4.1.9, which is also distributed with UML2Alloy.

The Alloy Analyzer works by translating the model into a boolean expression, which is analysed by SAT solvers embedded within the Alloy Analyzer. A user-specified scope on the model elements bounds the domain, making it possible to create finite boolean formulas for the evaluation by SAT solvers. The Alloy Analyzer offers two analysis methods. One is *simulation* and the other is *assertion checking*.

Simulation produces a random instance of the model that conforms to the constraints. The absence of an instance implies that most probably the model is inconsistent. It is possible from within the Analyser, to narrow down the statements, which produce the inconsistent model. This functionality is called UnSat Core [9].

Assertion checking, checks whether an assertion, which is a statement which should follow from the specification, is satisfied by the model. If it is not, a counterexample is presented to the user. A counterexample is an instance of the model, which violates the assertion.

This manual contains some information for both Alloy and the Alloy Analyzer for people familiar with UML but unfamiliar with Alloy. For more information on the Alloy Analyzer and the Alloy language, the reader is refereed to [5] and [1].

If the reader is interested in the transformation rules and the restrictions on the translation, the reader is referenced to Chapters 3 and 4. If the reader wants to learn details on how to use UML2Alloy, (s)he is referenced to Chapter 5. Finally Chapter 6 presents a number of examples to demonstrate how UML2Alloy can be used.

1.1 Minimum Requirements

UML2Alloy requires Sun Java 1.5

All the required third party libraries are distributed with UML2Alloy.

2 UML Profile for Alloy

In order to facilitate fully automated analysis of UML class diagrams enriched with OCL constraints, a UML profile for Alloy was developed. The purpose of the profile is two-fold. First, it does not allow the definition of UML models that can not be represented in Alloy (for example the UML standard allows for named elements (i.e. the *NamedElement* metaelement) without a name. Consequently it is possible to have an operation parameter without a name. On the other hand, Alloy requires that all predicate (the concept that maps to UML operations) parameters have a unique name.

Moreover the Alloy language has notions such as the *scope*, *simulation* and *assertion* commands, which allow fully automated analysis of Alloy models. On the other hand, the UML does not have such concepts. As a result, in order to achieve fully automated of UML models, through Alloy, the UML profile for Alloy, extends the UML with stereotypes and tagged values. Figure 2.1 depicts the stereotypes and the tagged values that may be specified. In the following we explain briefly the meaning of each stereotype.

2.1 Analysis

The *analysis* stereotype is used on UML packages that are going to be analysed using our method. A UML class diagram is required to have *exactly one* package stereotyped as *analysis*. The *analysis* stereotype defines three attributes (also called *tagged values*), the *defaultScope*, *intScope* and *time*. These tagged values are used during the transformation to set the default Alloy scope, the scope for integer numbers and the scope for the *time*[1] respectively. The *defaultScope*, *intScope* and *time* are positive integer numbers. Following Alloy's approach, if no *defaultScope* tagged value has been defined in the model, the default value is 3. Similarly if no default tagged value has been defined in the model for integer values, the scope for integer numbers is set to 4 and if no scope for the *time* has been defined, the scope for time is set to 3.

2.2 ScopedElement

Each class in a class diagram can be stereotyped as a *scopedElement*. The scopedElement stereotype defines a tagged value (*scope*), which is used to limit the number of instances of an element when a system instance is being checked by the SAT solver. This is used to override the *defaultScope* attribute of the *analysis* stereotype in order to define a different scope for the particular class on which the stereotype is applied.

[1] The scope for time is used only when modelling dynamic systems.

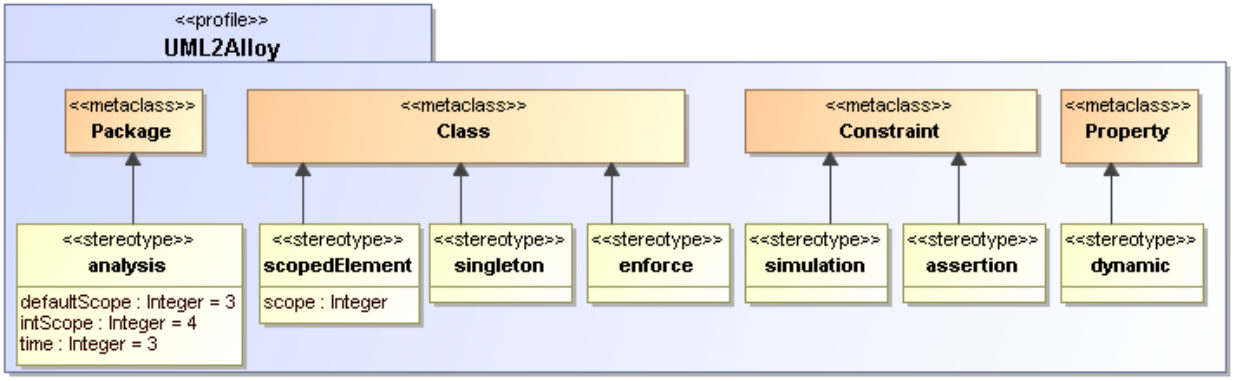

Figure 2.1: UML Profile for Alloy Stereotypes

2.3 Singleton

A *singleton* stereotype can be applied to a class and is used to define the well known singleton design pattern [3]. Classes annotated with this stereotype, are restricted to have *exactly* one instance in the model.

2.4 Enforce

In general an instance of a UML class diagram may be partial (i.e. some classes may not have any instances). This stereotype is used on classes that are required to have *at least one* instance during the analysis.

2.5 Simulation

In Alloy, a first-order logic statement can be used to simulate a model. This statement corresponds to the Alloy *run* command [5, Section 4.6]. Similarly in a UML class diagram an OCL constraint can be used to simulate the model. We use the *simulation* stereotype for this purpose. More specifically, an OCL statement stereotyped as simulation, will be automatically transformed to an Alloy simulation (*run*) command. An Alloy run command can be used with the Alloy Analyzer to create a random instance of the model that conforms to the statement and the constraints of the model.

2.6 Assertion

Similarly to simulation commands, *assertion* commands can be used to check if a statement that depicts a property of the system, is satisfied by the model. In a class diagram, an OCL statement can be used to depict an assertion. The *assertion* stereotype can be used on OCL statements, which will be transformed to Alloy assertions. It is important to note that OCL constraints annotated with the *simulation* or *assertion* stereotypes cannot have any pre and post conditions.

3 Transforming UML

This section provides a brief informal overview of the transformation rules from UML to Alloy. The rules are implemented in UML2Alloy. This section also mentions any restrictions that might exist.

3.1 Packages

◯ A UML Package is transformed to an Alloy module.

Restrictions

X Only one Package may be transformed to an Alloy module at a time.

X Package import is not supported.

3.2 Classes, Subclasses and Types

◯ Translates Classes to Signatures. If the class *isAbstract* it will be translated to an abstract Signature.

◯ Translates subclasses to subsignatures. The default subclassing semantics is {incomplete, disjoint} [8, p. 72]

◯ Enumerators are translated to abstract Signatures and Enumerator literals are translated to subsignatures with one element (Singletons). For more information see Section 3.3.

◯ Custom data types. We transform all Classes with the "DataType" stereotype to Signatures. Currently custom data types are not treated any different than any other Class. In the future we will extend the transformation to support the notion of Datatypes according to the UML standard. In particular, the UML standard specifies that Datatypes are identified by their value, rather than their object id.

Restrictions

X The only primitive types supported are *Integers*.

X Multiple packages are not supported. Translation of the classes of one package at a time takes place.

X Multiple inheritance is not allowed. It is possible to express diamond multiple inheritance in the future.

X Only subclasses with {incomplete, disjoint} semantics are allowed. Again the rules for the rest of the cases have been defined, but we haven't implemented them.

X Subclasses can not redefine attributes, operations. Thus attribute and operation names can not have the same name as the attribute and operation names of the superclasses.

X The NameSpace to which Classes can belong to is not restricted by the UML specification. Thus a Class can belong to the NameSpace of another Class (*inner Classes*). Our rules only support Classes which belong to the NameSpace of a Package.

3.3 Enumerations

◯ Enumerations are transformed to abstract Signatures.

◯ Enumeration literals are translated to Subsignatures, which extend the abstract Signature of the Enumeration. Since each enumeration literal represents only one value, it is declared as a Signature represented by one atom in Alloy.

Restrictions

X Enumerations have to be defined in the UML class diagram as Classes with an "enumeration" stereotype. Enumeration literals have to be defined as attributes of the Class. Attributes have to be of *void* type.

3.4 Associations

Please note that in this section the term *Association End* is used, to reference the ends of an association. Even though this term is used in the UML specification (for example see [8, p. 14] or [8, p. 40]), the UML metamodel refers to Association Ends as *Properties* [8, p. 29].

◯ All association ends *have* to have names. Association end names have to be unique.

◯ Unidirectional Associations: The navigable Association End appears as a field in the Signature to which the original Class the association belonged to, was translated.

◯ Bidirectional Associations: Association Ends appear as fields in both the Signatures that participate in the relation. An additional fact is added to the model, to denote that the relations are symmetric.

◯ Multiplicities: Depending on the multiplicity of the Association Ends, the necessary multiplicity facts will be added to the model.

◯ Custom multiplicities are supported.

Restrictions

X Association End names have to be *unique* throughout the model (i.e. all Association Ends have to have unique names, in contrast with the UML specification, which allows two separate Association Ends to have the same name, as long as they belong to different Classes).

X Association End annotations are not currently supported (e.g. *subsets*, *ordered* etc.).

X Only binary associations are supported in the current version of the implementation.

X The UML specification allows for unspecified navigability [8, p. 41]. In UML2Alloy, an Association End needs to be either navigable or non-navigable.

X Aggregation and Composition are not supported. Associations with composition or aggregation have to be refactored using the method defined by Gogolla and Richters [4].

X Association Classes are not currently supported.

3.5 Class Attributes

◯ Class attributes are translated to Signature fields.

Restrictions

X Attributes have to be of a specific type (i.e. not *void*). This restriction does *not* apply for enumeration literals, see Section 3.3.

X Only attributes with multiplicity of 1 are supported.

3.6 Class Methods

◯ Class methods are transformed to Alloy predicates. For more information see Section 4.2.

Restrictions

X Class methods can not return any type, other than Boolean or Void.

X Operations in UML may raise exceptions. UML2Alloy does not deal with the exceptions.

4 Transforming OCL

This section presents the details on the translation of OCL statements (invariants and pre and postconditions) to Alloy.

IMPORTANT NOTE: NO shorthand notation is supported for OCL expressions. For example:
Use: collection → forAll(v:Type \| boolean-expression-with-v)
Do NOT use: collection → forAll(v \| boolean-expression-with-v)
Do NOT use: collection → forAll(boolean-expression)

Use: self.attribute
Do NOT use: attribute

4.1 Invariants

Please NOTE: Invariants stereotyped as Simulation or Assertion cannot reference the special variable 'self'. For simulation and assertion commands the expanded form of invariants should be used. For example instead of the statement:
context C1
'self.attribute = value'

the statement:

context C1
C1.allInstances → forAll(v1:C1 \| v1.attribute = value)

should be used.

- ◯ Class invariants are generally transformed to Alloy facts. They can also be transformed to Simulation or Assertions commands. This is defined during the transformation process. For more information please see the usage manual in Chapter 5.
- ◯ Each invariant is translated to a predicate and then a fact statement, which references the predicate is added.

Restrictions

- X Only Class invariants defined in the namespace of a Class are supported (i.e. Class invariants).
- X Invariants have to have a name. The name of each invariant needs to be unique throughout the model.

4.2 Operations

The effects of Operations is defined using OCL *pre* and *post* conditions.

NOTE: The current version of the implementation only supports postconditions.

◯ Operations are transformed to Alloy Predicates. Operations are defined in the context of a Class. Because in Alloy Predicates belong to the NameSpace of the model, an instance of the Signature on which the specification of the Predicate is applied, is passed as a parameter to the Class. Therefore all references to the *self* keyword used in OCL are transformed to the parameter passed in the Predicate. For a detailed example please see Section 6.3.

Restrictions

X The current implementation supports one OCL expression as specification to the Operation.

X Pre and Post conditions have to have a name.

X Operations can be of either Boolean or Void type.

X Currently Operations can not have any parameters.

4.3 OperationCallExps

In OCL a large number of expressions are OperationCall expressions. For example *equality*, *implies*, *and*, *or*, etc. expressions, are considered to be OperationCall expressions. The following shows how those OperationCall expressions are transformed into Alloy.

4.3.1 And/Or

◯ An OCL *and* OperationCall is transformed to an Alloy binary formula (*BinaryFormula*) with an AND Logical operator.

◯ Similarly an OCL *or* OperationCall is transformed to an Alloy binary formula, with an OR Logical operator.

4.3.2 Dot

Dot OperationCall expressions in OCL, are navigation expressions, which use the navigation dot (.).

◯ An OCL *Dot* OperationCall is transformed to an Alloy binary expression *BinaryExpr* with a JOIN_DOT binary expression Operation.

4.4 If

◯ An OCL *If Expression* is transformed to an Alloy implies formula (*ImplicationFormula*). In Alloy an ImplicationFormula can have an *else* clause, which is what the OCL *else* clause is translated to.

4.4.1 Implies

◯ Implies OperationCalls are directly transformed to an Alloy implies formula (*ImplicationFormula*).

4.5 Iterator Expressions

◯ An OCL Iterator Expression is transformed to an Alloy *Quantified Formula*.

Restrictions

X Currently only the *forAll* and *exists* Iterator Expressions are supported.

4.6 notEmpty/isEmpty

- ◯ An OCL notEmpty() expression is transformed to an Alloy *QuantifiedExpr* with the *some* quantifier. In particular it is transformed to an Alloy Formula, which enforces the existence of some elements.
- ◯ Similarly an OCL isEmpty() expression is transformed to an Alloy *QuantifiedExpr* with the *no* quantifier.

4.7 size

- ◯ An OCL size() Operationcall is transformed to an Alloy *CardinalityExpr*.

4.8 union

- ◯ An OCL union() Operationcall is transformed to an Alloy *BinaryExpr* with a union operation.

4.9 Including/Excluding

- ◯ An OCL *including()* Operationcall is transformed to an Alloy *BinaryExpr* with a union operation (according to Table A.4 of OCL 2.0 specification [6] the *including* set-theoretic interpretation is the same as the union set-theoretic operation).
- ◯ An OCL *excluding()* Operationcall is transformed to an Alloy *BinaryExpr* with a set difference operation (according to Table A.4 of OCL 2.0 specification [6] the *excluding* set-theoretic interpretation is that of the set difference set-theoretic opearation).

4.10 Select

- ◯ An OCL *select()* Operationcall is transformed to an Alloy *ComprehensionExpr*.

4.11 Includes/IncludesAll

- ◯ OCL *includes()* and *includesAll()* Operationcalls are transformed to an Alloy *Formula* with a *subsets* compare operation. According to Table A.3 of OCL 2.0 specification [6] *includes* and *includesAll* set-theoretic interpretation is that of the 'subsets' set-theoretic operation.

4.12 Excludes/ExcludesAll

- ◯ OCL *excludes()* and *excludesAll()* Operationcalls are transformed to an Alloy *Formula* with a negation of the subsets compare operation. According to Table A.3 of OCL 2.0 specification [6] *excludes* and *excludesAll* set-theoretic interpretation is that of the the negation of the 'subsets' set-theoretic operation.

4.13 Not

- ◯ OCL *not* expressions (expressions with a negation) are directly transformed to Alloy Negation Formulas *NegFormula*.

4.14 Variable Expressions

- ◯ OCL *Variable Expressions* (for example in the expression '*self.name*', *self* is a Variable Expression), are transformed to Alloy Variable Expressions (*VariableExpr*).

4.15 Integer Expressions

◯ UML Class attributes of type *integer* are transformed to fields of type *Int*.

◯ OCL *LiteralIntegers* [7, p.60] are transformed to Alloy *LiteralIntExprs*.

Restrictions

X Currently (version 0.5) do not support operations (i.e. addition, subtraction, multiplication, division) over integer values.

4.16 oclIsKindOf

◯ An oclIsKindOf OCL statement is transformed to Alloy using the set containment operator.

Restrictions

X The oclIsKindOf can only be applied on user defined classes. It cannot be applied on UML predefined types (i.e. 10.oclIsTypeOf(Integer) is not allowed).

4.17 Boolean Literal

◯ The Boolean literal *true* is transformed to an Alloy expression that always evaluates to true (for example '*1=1*'). Similarly the Boolean literal *false* is transformed to an Alloy expression that always evaluates to false (for example '*1!=1*').

5 UML2Alloy: Usage Manual

This chapter is a brief guide on how to use UML2Alloy. It is suggested that ArgoUML version 0.19.5 is used to develop any models. The reason for this is that ArgoUML provides syntax checking for OCL statements. **Please note that apparently some versions (since ArgoUML version 0.20) have a bug in the OCL syntax checker and as a result it is strongly recommended to use version 0.19.5**[1].

5.1 Installation

Download the zip file of the distribution and unpack it on your local storage, ensuring the directory structure of the contents of the zip file is preserved. In the command prompt navigate to the directory where UML2Alloy was unzipped and use the *run.bat* if you are on windows, or the *run.sh* file if you are using *nix. If you are on a *nix, operating system, you will have to give execution rights to the file *run.sh*. Please run UML2Alloy from the command prompt, because for the time being a number of error messages and warnings print to the console.

5.2 Using UML2Alloy

Figure 5.1 shows the main screen UML2Alloy. On the right hand side of the screen you can see the messages log. In the first tab you need to enter the path to the XMI file that contains the UML model that will be transformed to Alloy. You also need to specify the path of the Alloy text file that will be generated if the transformation is successful.

5.2.1 Scopes Tab

After you enter the path of the XMI file to parse and the Alloy file to generate, you press the '*Parse XMI*' button. This will attempt to parse the XMI file. If the parse completes successfully, the other tabs in the user interface are enabled. In the '*scopes*' tab you may specify the scopes of the elements. Figure 5.2 depicts a screenshot of this tab.

On the top of the screen of the '*scopes*' tab, the user may specify the '*default scope*', which is the scope that will be applied to all model elements. This can be overridden for specific model elements if a value other than -1 exists in the *scope* box of a specific model element. By checking the checkbox of the last column the user can *enforce* that a model element will have instances in the model. For a concrete example on the usage of this function, please refer to the examples in Chapter 6.

[1] ArgoUML 0.19.5 can be downloaded from: http://argouml-downloads.tigris.org/argouml-0.19.5/

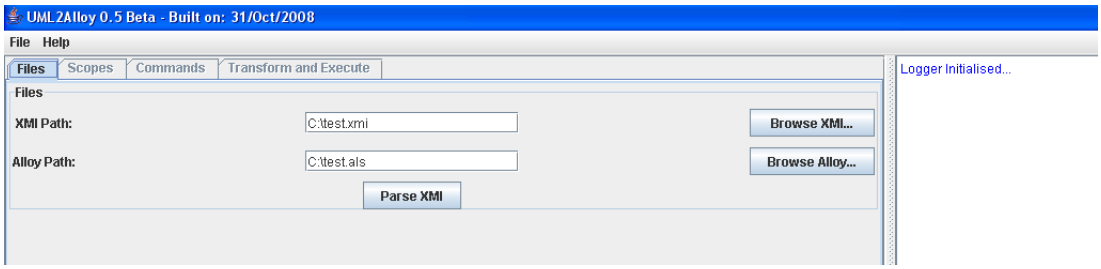

Figure 5.1: Files Tab of UML2Alloy

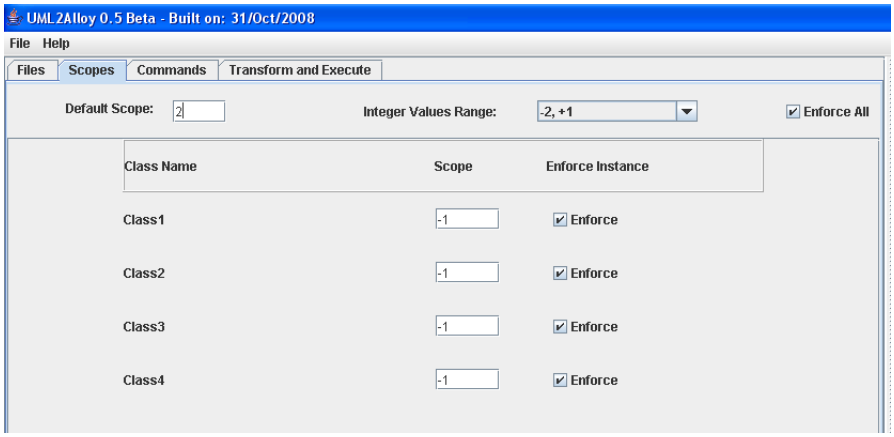

Figure 5.2: Scopes Tab of UML2Alloy

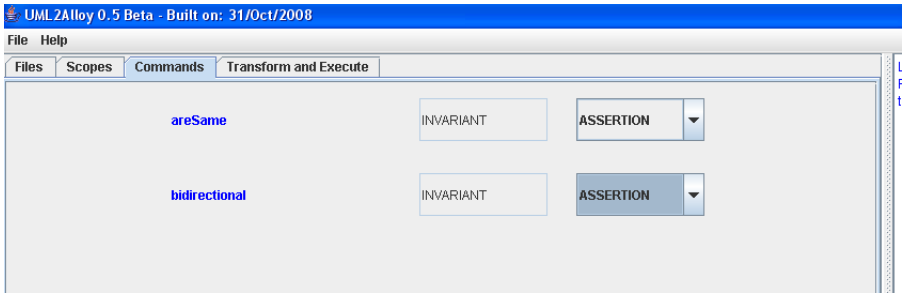

Figure 5.3: Commands Tab of UML2Alloy

5.2.2 Commands Tab

In the '*Commands*' tab (Figure 5.3) the user is presented with all the Invariants and Operations of the UML model. The first column depicts the Operation or Invariant name. Please note that by clicking on the name of Operation or Invariant, you can see its original OCL specification. The second column shows if the statement is an Invariant or Operation in the original UML model. Finally the third column has a drop down menu, which allows the user to select to what kind of Alloy statement the original UML statement will be transformed.

The drop down menu with the selections is populated with the following rules. An OCL Invariant can be translated to an Invariant (*Fact* in Alloy terms), *Assertion* or *Simulation* statement.

5.2.3 Transform and Execute

Figure 5.4 shows a screenshot of the *Transform and Execute* tab. Once the user has set the desired parameters in the previous tabs, (s)he can press the *Transform2Alloy* button, that will carry out the actual transformation of the UML model into Alloy. If the transformation is successful the Alloy model will be generated in the path that was specified by the use in the '*Files*' tab.

When the transformation to Alloy is finished, it is possible to select the command to execute and execute it from within UML2Alloy, using the Alloy Analyzer API. In particular, the '*Command2Execute*' drop down box will be populated with the commands in the generated Alloy model, as shown in Figure 5.5.

The user can select the command to execute from the drop down menu and hit the '*Execute*' button. If the execution is successful, the result will be displayed in the '*Alloy execution log*'. If the execution returns an instance (a random instance in the case of simulation or a counterexample in the case of assertions), the XML output of the instance will be displayed in the Alloy execution log. It is also possible to click on the '*View Execution Result (Graph)*' button, that will show the instance using

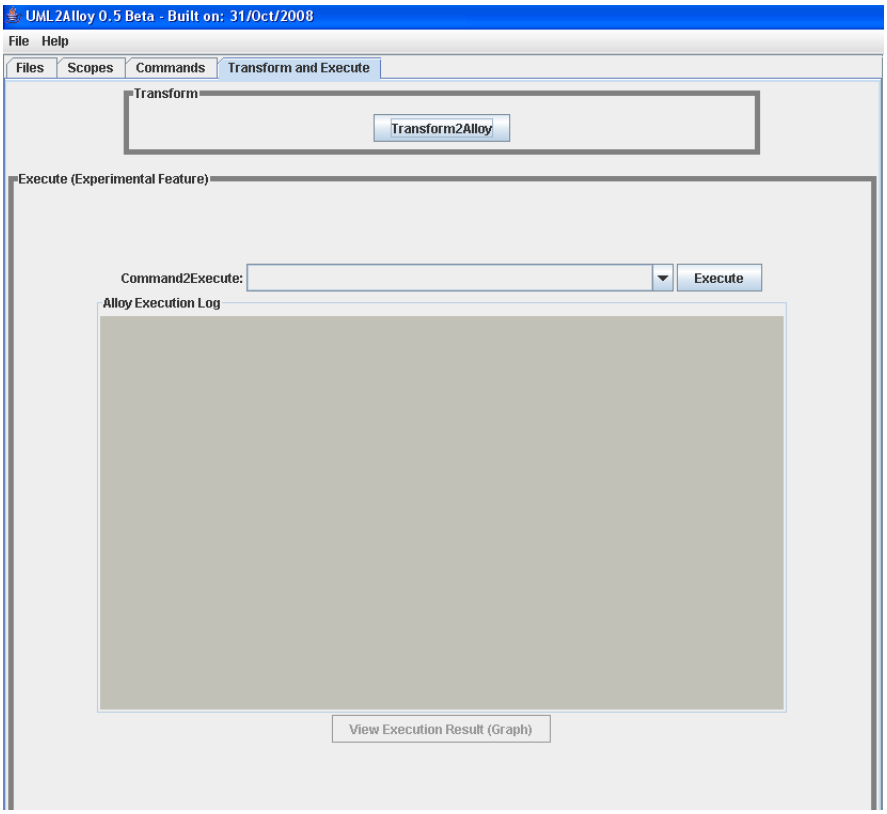

Figure 5.4: Transform Tab of UML2Alloy

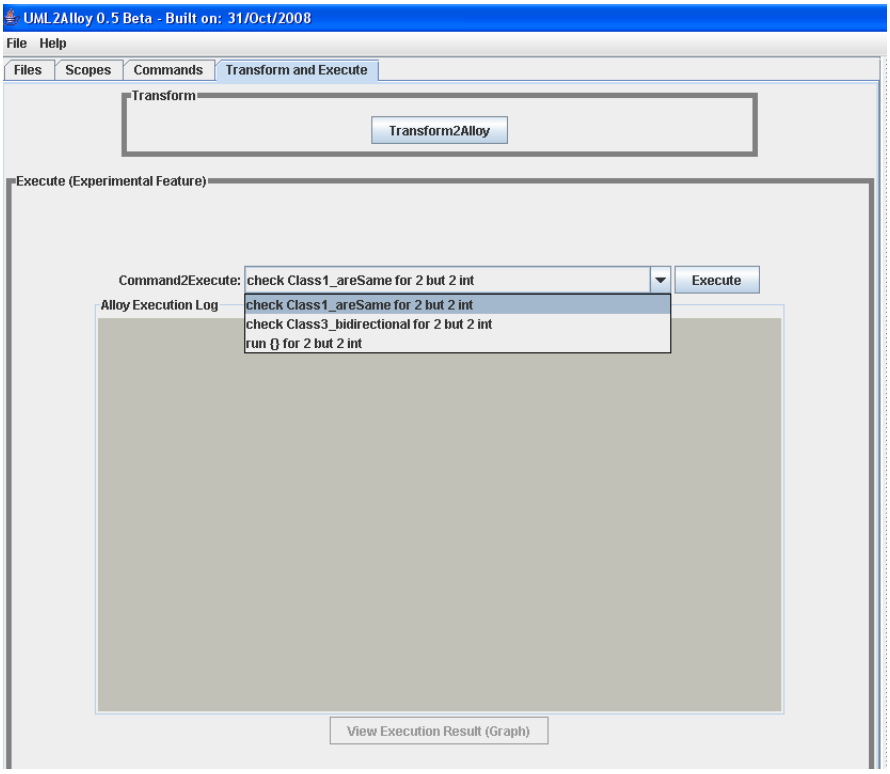
UML2Alloy 0.5 Beta - Built on: 31/Oct/2008
File Help
Files Scopes Commands Transform and Execute
Transform
Transform2Alloy
Execute (Experimental Feature)
Command2Execute:
check Class1_areSame for 2 but 2 int
Execute
Alloy Execution Log
check Class1_areSame for 2 but 2 int
check Class3_bidirectional for 2 but 2 int
run {} for 2 but 2 int
View Execution Result (Graph)

Figure 5.5: Selecting the Command to Execute

Alloy's graphical engine.In a future release of UML2Alloy the instance will be shown as a UML object diagram.

If an instance cannot be found the UnSat Core will be queried and the original UML and OCL elements responsible for the inconsistency will be displayed. Please note that in the case of an assertion not producing a counterexample the UnSat core can represent the original UML and OCL elements responsible for the validity of the assertion.

6 Examples

This chapter presents a number of simple examples to demonstrate the usage of UML2Alloy. **Please note the ArgoUML project files as well as the generated Alloy models can be found under the '*Examples*' section of the UML2Alloy website.**

Section 6.1 describes a very simple example, which demonstrates how counterexamples to assertion statements can be acquired and how the UnSat Core of assertions can be interpreted.

6.1 Two Unidirectional vs One Bidirectional Associations

In this example we want to assert if two unidirectional associations between two classes have the same meaning as one bidirectional association. To do so, we generate the model depicted in Figure 6.1. The model was developed using ArgoUML version 0.19.5.

The model does not have any constraints. We want to assert that the two unidirectional associations between Class1 and Class2 have the same effect on the model as the bidirectional association between Class3 and Class4. The assertion we use is stated in OCL:

```
-- We call this statement 'areSame'
context Class1 inv areSame :
Class1 . allInstances() -> forAll (
cl1: Class1 | cl1 . c2 . c1 = cl1 )
```

This assertion states that for each Class1, if we navigate to Class2 and come back we will end up in the same object as where we started.

In ArgoUML we define this assertion as an *Invariant* and when we use UML2Alloy in the '*Commands*' tab, we define that we want to transform it as an assertion, i.e. a statement to check. Another option would be to stereotype the statement as *assertion*.

Similarly we formulate the following OCL statement to reason about the association between Class3 and Class4.

```
-- We call this statement 'bidirectional'
    context Class3
    inv bidirectiional : Class3 . allInstances -> forAll (
    cl3 : Class3 | cl3 . c4 . c3 = cl3 )
```

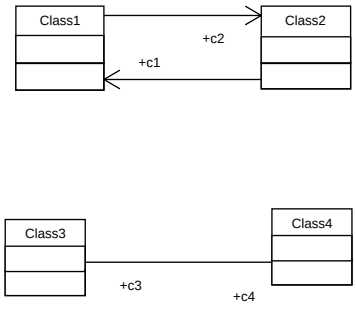

Figure 6.1: Example UML model of two classes connected with two unidirectional and one bidirectional association

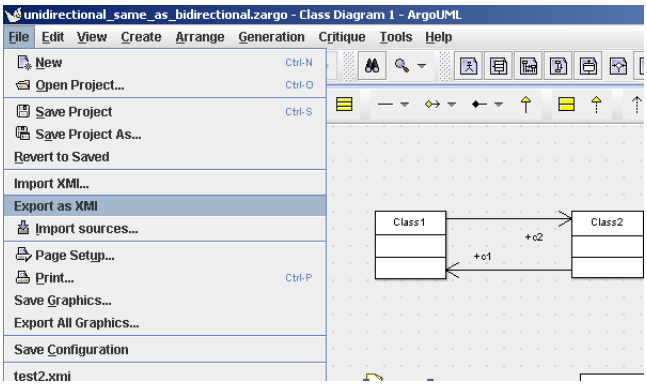

Figure 6.2: Exporting model to XMI using ArgoUML

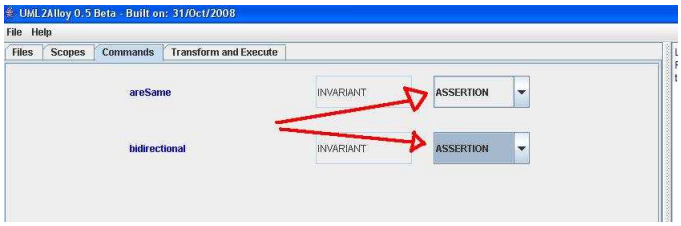

Figure 6.3: Selecting that the areSame and bidirectional statements are Assertions

We now export the model to XMI format from within ArgoUML. Figure 6.2 depicts a screenshot that shows how to achieve that in ArgoUML.

Next we invoke UML2Alloy. In the first tab we select the path to the XMI file and the Alloy file we want to create. We then click on the '*Parse XMI*' button.

In the next tab we select the scopes for each model element (Figure 5.2). We specify a default scope of 2. This means that the Alloy Analyzer will probe to find a counterexample to the assertion, using up to two atoms (i.e. instances) for each class. We also check the *enforce* checkbox on all classes, to enforce that at least one instance of each class will exist in the solution. Finally we do not change the '*Integer Values Range*' combobox, because our model does not use any integers.

In the *Commands* tab (Figure 5.3) we specify how each OCL statement will be mapped to Alloy. We want to check if both the *areSame* and *bidirectional* OCL statements are satisfied. They therefore need to be transformed to assertions. This is specified by selecting in the drop down menus that we want the original OCL invariants to be transformed to assertions (indicated by the red arrows in Figure 6.3).

We now move to the '*Transform and Execute*' tab and click on the '*Transform2Alloy*' button. Once the transformation is successful, the *Command2Execute* drop down menu will be populated with the available commands to execute (as in Figure 6.4). In this case the available commands are the *areSame* and *bidirectional* assertions. Moreover by default a simulation command is available to execute ('*run* {} *for...*' in the drop down menu). The simulation command will attempt to find an instance of the model that conforms the constraints. Next to the name of each command are the scopes. For this example, the Alloy Analyzer will attempt to find a counterexample to the assertions for 4 elements, but for a bitwidth of 2 for integer numbers.

We first select from the drop down menu the '*areSame*' assertion and press on *Execute*. The Alloy Analyzer produces a *counterexample*. The counterexample in XML format can be seen in the '*Alloy Execution Log*' part of the screen. We can click on the '*View Execution Result (Graph)*' to view the counterexample, using Alloy Analyzer's visualisation engine. The counterexample using Alloy's visualisation engine is depicted in Figure 6.5.

From a closer examination of the instance we can see that there is a class, *Class1_0* related to *Class2_0*,

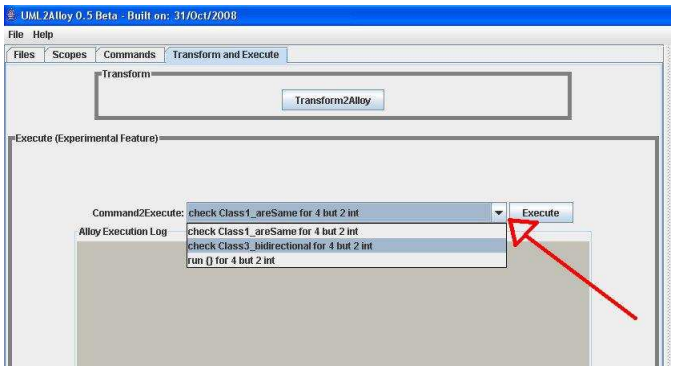
UML2Alloy 0.5 Beta - Built on: 31/Oct/2008
File Help
Files Scopes Commands Transform and Execute
Transform
Transform2Alloy
Execute (Experimental Feature)
Command2Execute:
check Class1_areSame for 4 but 2 int
Execute
Alloy Execution Log
check Class1_areSame for 4 but 2 int
check Class3_bidirectional for 4 but 2 int
run {} for 4 but 2 int

Figure 6.4: Available Commands to Execute

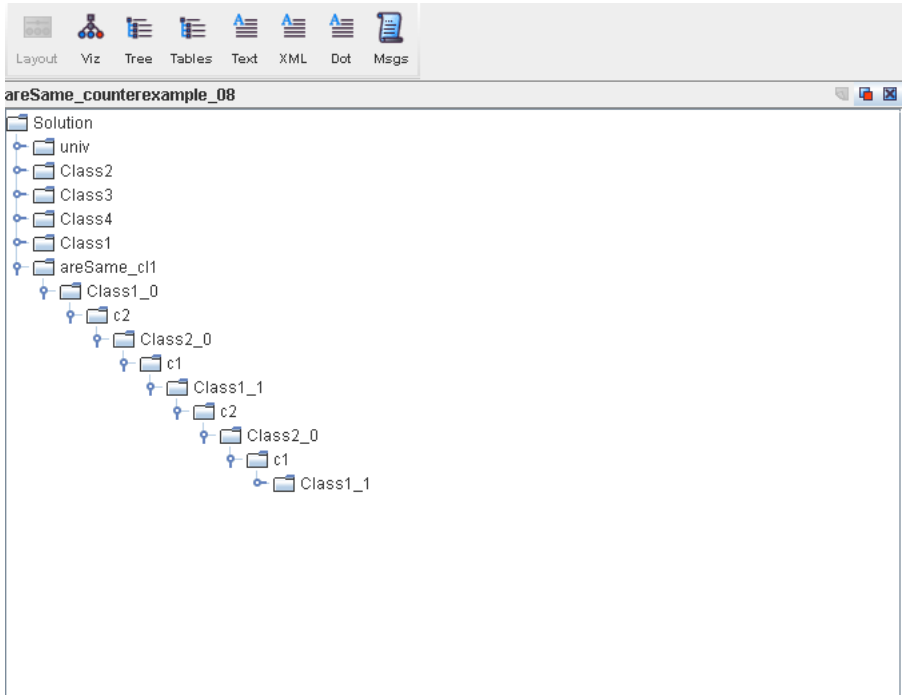

Figure 6.5: Counterexample

which in turn is related to *Class1_1*. The assertion therefore fails, because navigating from a class to another class and back, we end up with a different class. Currently the results of the analysis are represented using the Alloy Analyzer representation form (for example as a Tree, as shown in Figure 6.5). In the future we plan to represent the results of the analysis in terms of UML (i.e. object diagrams).

Checking the *bidirectional* assertion does not provide any counterexamples.

6.2 Precise UML for solving Sudoku

In this example, we want to solve a simple sudoku puzzle using UML. Figure 6.6 depicts a 6x6 sample sudoku puzzle. It is a straight procedure to model this puzzle in UML. Figure 6.7 shows the UML class diagram that represents the puzzle. This puzzle consists of 36 cells. Each cell has a *rowIndex*, which represents the row of the cell, a *columnIndex*, which represents the column of the cell and a *value*, which represents the contents of the cell. Finally, a cell belongs to a *region* and each region is represented by an integer number [1].

[1] The sample puzzle of Figure 6.6 has 6 regions. The region borders are designated by the bold lines.

		6	3		2
3	2				6
				2	3
2	5				
5				3	1
1		2	4		

Figure 6.6: A Sample Sudoku Puzzle

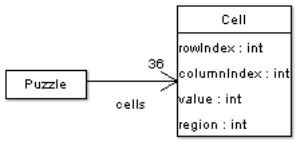

Figure 6.7: A Class Diagram representing the Sudoku puzzle

OCL is used to specify the constraints of the puzzle and the numbers that exist already in the puzzle. The following OCL statement depicts the rules of the puzzle:

```
context Cell
inv rules :
Cell.allInstances() -> forAll (c1, c2: Cell |
( ( c1.rowIndex = c2.rowIndex or
      c1.columnIndex = c2.columnIndex or
      c1.region = c2.region)  and
  ( c1 <> c2 ))  implies
    c1.value <> c2.value)
```

The OCL statement requires that for every two cells, *c1* and *c2*, if they belong to the same row or column or region, and they are not the same cell, then the numbers of *c1* and *c2* are different. The complete UML model and OCL constraints of this puzzle and the generated Alloy model can be found in appendix B.

UML2Alloy was used to transform the UML representation of the puzzle to Alloy. The Analyzer provided a solution to the puzzle [2]. The solution the Analyzer found is illustrated in Figure 6.8.

6.3 Man-in-the-middle

This section presents an example of how the man-in-the-middle attack can be analysed using Alloy, through UML2Alloy. Please note that in these example we do *not* care about the order of the messages exchanged between the parties involved, but only about which party has access to which information. This is similar to the approach taken by Torlak et al. [10]. Since no attribute or association end is stereotyped as '*dynamic*' the analysis is carried out over a single state (i.e. the operation pre and post conditions are dealt as invariants). First the unprotected (plaintext) version of the attack is presented in the next section.

[2] Please note that executing this puzzle requires additional memory. As a result, when launching UML2Alloy from the command prompt use the '*-Xmx*' java switch to provide additional memory to the Java VM. For example use the command: '*java -jar -Xmx256m uml2alloy.jar*'.

4	1	6	3	5	2
3	2	5	1	4	6
6	4	1	5	2	3
2	5	3	6	1	4
5	6	4	2	3	1
1	3	2	4	6	5

Figure 6.8: Solution of the sample Puzzle of Figure 6.6

6.3.1 Unprotected

Figure 6.9 shows the class diagram of the unprotected man in the middle attack. The *Client* is connected through an *Attacker* to the *LoginManager*. The Client uses a *username* and a *password* to login to the *LoginManager*. If the correct username and password are supplied the Client has access to the personalised *homePage* otherwise he has access to a *visitorPage*.

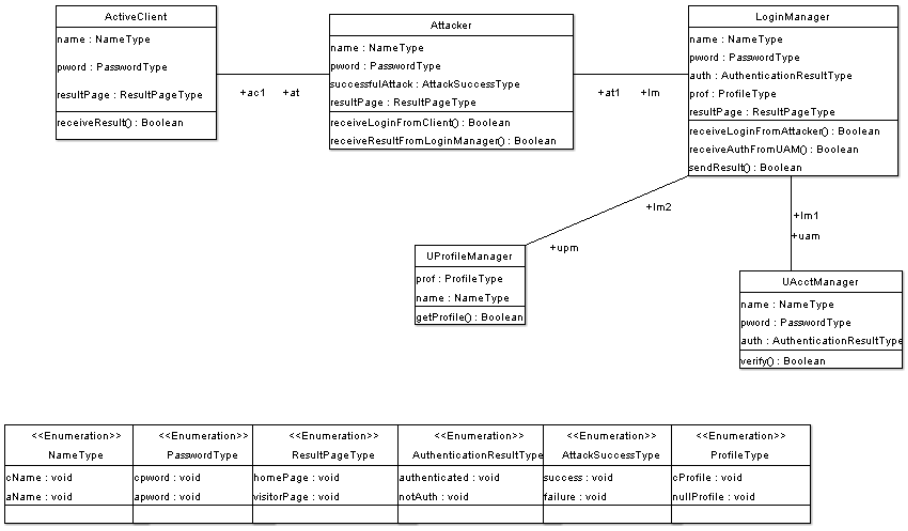

Figure 6.9: Man in the Middle Class Diagram (Unprotected)

In this example we check that if the Attacker eavesdrops on the communication, whether (s)he can have access to the homepage. This model has one invariant, called *exec_main*. This invariant is used to simulate the model. Also there are two assertions, *assert1* and *assert2*, which need to be transformed to Alloy assertions. The specification of *Assert1* is the following:

```
all a: ActiveClient | a.at.resultPage = visitorPage
```

Assert1 asks whether all Attackers in the system will have access to the visitorPage (i.e. no Attacker has access to the *homePage*). Alloy presents a counterexample with an Attacker having access to the *homePage* using the credentials of the *ActiveClient*.

6.3.2 TLS Protected

Figure 6.10 shows the class diagram of the man-in-the-middle attack over the TLS protocol. In this case no username or password is used by the client to gain access to the homepage. Instead, the TLS protocol utilises asymmetric cryptography, which means that the server needs to have a certificate signed by an authority.

Figure 6.10: Man in the Middle Class Diagram (TLS)

This example was analysed for a scope of 8. The Alloy Analyzer produced an instance of the model where the Attacker can view the homePage. For a close inspection of the instance, it became apparent that this happens if the Attacker has access to the encrypted public key of the client. In practice this can only happen if the Attacker has access to the Client's private key. Assuming that the Client keeps his private key safe, we added the following invariant in the model:

```
context ActiveClient
inv: ActiveClient.allInstances() ->
forAll(ac:ActiveClient | ac.at.encryptedAPublicKey <>
ac.encryptedCPublicKey)
```

After the invariant was added to the model, using a scope of 20 the Alloy Analyzer did not produce any counterexample.

7 Developer API

This section contains information on how to use the developer API of UML2Alloy to carry out an automated transformation of a UML model that conforms to the UML profile for Alloy, into Alloy.

UML2Alloy comes with a simplified version of the UML metamodel. The classes of this metamodel can be found in the package: '*org.uml2alloy.umlmmsimplifiedprofile*'. Figure 7.1 depicts a UML model of the classes of the package. In order to carry out the transformation, first we need to use the classes in the package to define a UML model. The following section shows how this is achieved.

7.1 Defining a UML Model

Assume the UML model of Figure 7.2. The model has the following constraints:

```
context Person
inv ageInv: self.age > 0

-- The following constraint is stereotyped as 'simulation'
context Person inv adultSimulation:
Person.allInstances() -> forAll (p:Person | p.age > 18)

-- The following constraint is stereotyped as 'assertion'
context Person
inv anAssertion: Person.allInstances() -> forAll(p:Person | p.wife -> notEmpty() implies p.wife.husband = p)
```

This model can be represented with the following Java code:

```
1  // Imports
2  package org.uml2alloy.tests;
3
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6
7  import org.uml2alloy.AbstractUML2AlloyTransformer;
8  import org.uml2alloy.UML2AlloyTransformer;
9  import org.uml2alloy.alloy4.AlloyModel;
10 import org.uml2alloy.sitra4uml2alloy.RuleNotFoundException;
11 import org.uml2alloy.transformer.exceptions.UML2AlloyTransformationException;
12 import org.uml2alloy.umlmmsimplifiedprofile.Association;
13 import org.uml2alloy.umlmmsimplifiedprofile.AssociationImpl;
14 import org.uml2alloy.umlmmsimplifiedprofile.Class_;
15 import org.uml2alloy.umlmmsimplifiedprofile.Class_Impl;
16 import org.uml2alloy.umlmmsimplifiedprofile.Constraint;
17 import org.uml2alloy.umlmmsimplifiedprofile.ConstraintImpl;
18 import org.uml2alloy.umlmmsimplifiedprofile.GeneralizationImpl;
19 import org.uml2alloy.umlmmsimplifiedprofile.PackageImpl;
20 import org.uml2alloy.umlmmsimplifiedprofile.Package_;
21 import org.uml2alloy.umlmmsimplifiedprofile.PrimitiveType;
22 import org.uml2alloy.umlmmsimplifiedprofile.PrimitiveTypeImpl;
23 import org.uml2alloy.umlmmsimplifiedprofile.Property;
24 import org.uml2alloy.umlmmsimplifiedprofile.PropertyImpl;
25 import org.uml2alloy.umlmmsimplifiedprofile.stereotypes.AnalysisStereotype;
26 import org.uml2alloy.umlmmsimplifiedprofile.stereotypes.AssertionStereotype;
27 import org.uml2alloy.umlmmsimplifiedprofile.stereotypes.EnforceStereotype;
28 import org.uml2alloy.umlmmsimplifiedprofile.stereotypes.ScopedElementStereotype;
29 import org.uml2alloy.umlmmsimplifiedprofile.stereotypes.SimulationStereotype;
30
31 // CODE
32
33 // Define the package called 'Example'.
34                 Package_ p  = new PackageImpl("Example");
35                 // The package is stereotyped as 'analysis'. Set the tagged values
36                 // and add an instance of the stereotype to the package
37                 AnalysisStereotype.setDefaultScope_(4);
38                 AnalysisStereotype.setIntScope_(6);
39                 p.addStereotypes(AnalysisStereotype.getInstance());
40
41                 // Add the integer primitive type and add it to the package
42                 PrimitiveType ptInt = PrimitiveTypeImpl.buildIntegerPrimitiveType();
```

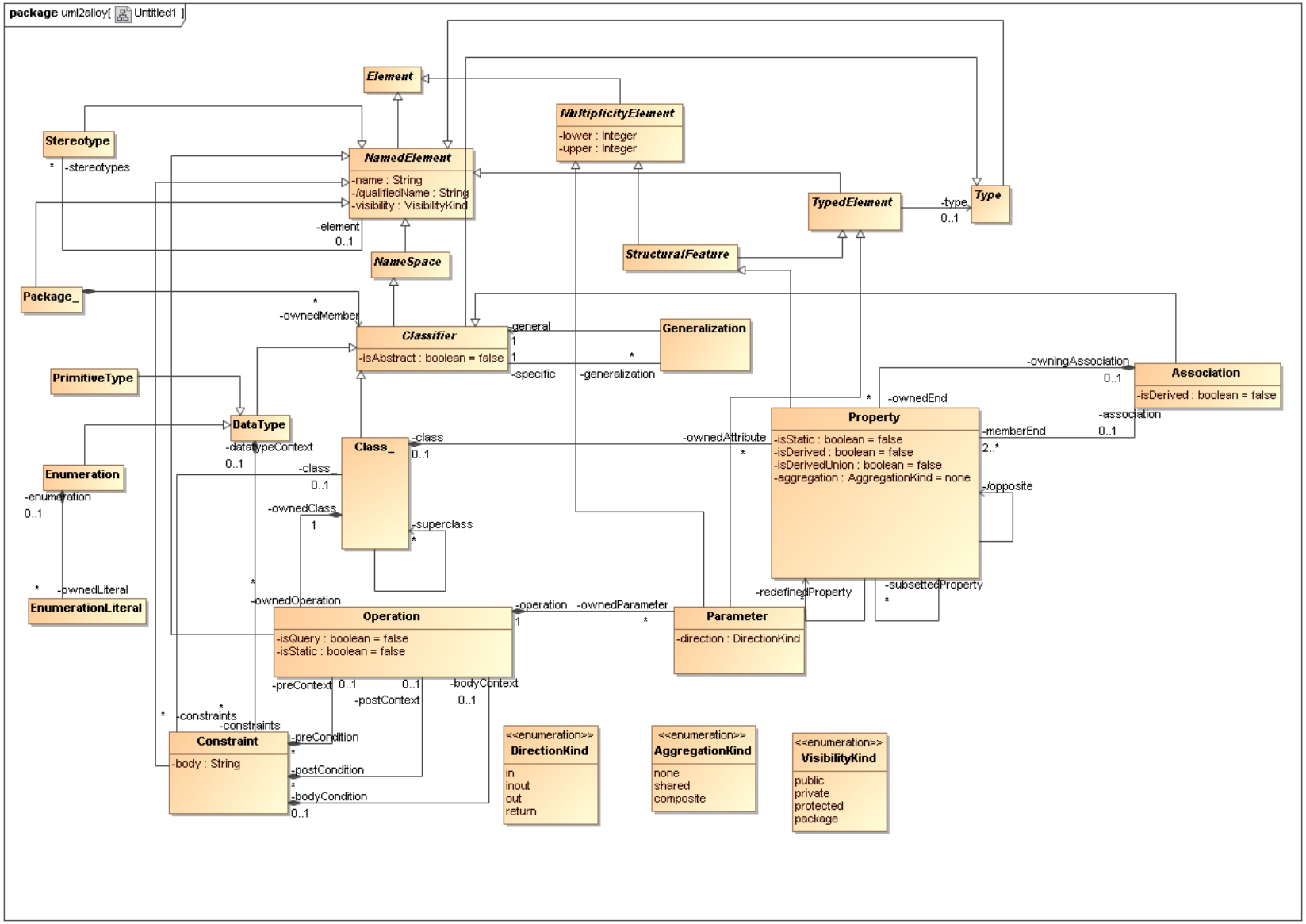

Figure 7.1: Package 'org.uml2alloy.umlmmsimplifiedprofile'

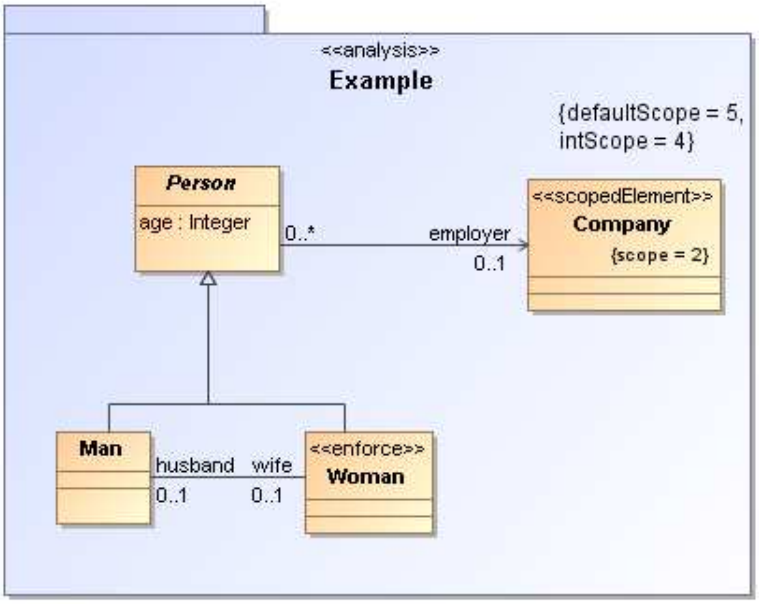

Figure 7.2: Example UML Model

```
43              p.addOwnedMember(ptInt);

45              // Add class Person, make it abstract and add it to the package
46              Class_ person = new Class_Impl("Person");
47              person.setIsAbstract(true);
48              p.addOwnedMember(person);

50              // Add class Man
51              Class_ man = new Class_Impl("Man");
52              p.addOwnedMember(man);

54              // Add class Woman. Also add the
55              // 'enforce' stereotype to the woman
56              Class_ woman = new Class_Impl("Woman");
57              woman.addStereotypes(EnforceStereotype.getInstance());
58              p.addOwnedMember(woman);

60              // Add class Company.
61              // Also add the scopedElement Stereotype
62              Class_ company = new Class_Impl("Company");
63              company.addStereotypes(new ScopedElementStereotype(2));
64              p.addOwnedMember(company);

66              // Add the generalization between Person and Man
67              new GeneralizationImpl(man,person);
68              // Add the generalization between Person and Woman
69              new GeneralizationImpl(woman,person);

71              // Add the 'age' attribute of Person
72              Property age = new PropertyImpl("age",ptInt);
73              person.addOwnedAttribute(age);

75              // Add the 'employer' association end
76              Property employer = new PropertyImpl("employer",company);
77              employer.setLower(0); // Lower multiplicity of zero
78              person.addOwnedAttribute(employer);

80              // The non-navigavle association end
81              // of the association between Person and Company
82              Property person_non_navigable = new PropertyImpl("person_non_navigable", person);
83              person_non_navigable.setLower(0);
84              person_non_navigable.setUpper(-1);

86              // Association between Person----> Company
87              Association asso_person_company = new AssociationImpl(employer,person_non_navigable);
88              // Person_non_navigable association end is not navigable
89              // therefore it belongs to the association, rather than a class
```

```
90              asso_person_company.addOwnedEnd(person_non_navigable);

92              // Add association to the package owned elements
93              p.addOwnedMember(asso_person_company);

95              // The wife association end
96              Property wife = new PropertyImpl("wife", woman,0,1);
97              man.addOwnedAttribute(wife);

99              // The husband association end
100             Property husband = new PropertyImpl("husband", man, 0, 1);
101             woman.addOwnedAttribute(husband);

103             //         Association between Man <----> Woman
104             Association asso_man_woman = new AssociationImpl(wife, husband);

106             p.addOwnedMember(asso_man_woman);

108             // Also add invariants
109             Constraint ageInv = new ConstraintImpl("ageInv",
110                             "context Person inv ageInv: self.age > 0");
111             person.addConstraints(ageInv);

113             // Also add the ageSim Simulation Invariant
114             Constraint ageSim = new ConstraintImpl("ageSim",
115             "context Person inv ageSim: Person.allInstances() -> forAll (p:Person | p.age > 18)");
116             ageSim.addStereotypes(SimulationStereotype.getInstance());
117             person.addConstraints(ageSim);

119             // Finally add an assertion
120             Constraint anAssertion = new ConstraintImpl("anAssertion",
121                     "context Person inv anAssertion: Person.allInstances() ->
122                     forAll(p:Person | p.wife -> notEmpty() implies p.wife.husband = p)");
123             anAssertion.addStereotypes(AssertionStereotype.getInstance());
124             person.addConstraints(anAssertion);
```

7.2 Transforming

To carry out the actual transformation the *UML2AlloyTransformer* class needs to be used. For a detailed example, please see the UML2Alloy website, under the 'examples' section.

Appendices

Appendix A XMI Compatibility

We use the KMF XMI parser, so our parser is compatible with the XMI files the KMF parser can read. We have tested UML2Alloy with a number of UML design CASE tools. Table A.1 shows the results of our tests.

CASE TOOL	VERSION	SUCCESS
ArgoUML	0.19	YES
ArgoUML	0.22	YES
Poseidon For UML	4.1 Community Edition	YES
MagicDraw	9.5 Community Edition	YES
MagicDraw	Any Version >= 10	NO

Table A.1: CASE Tools Support

Appendix B Complete UML Model of Sudoku Puzzle

Please note that the ArgoUML project, the XMI file and the generated Alloy file of this example, can be found on the UML2Alloy website, under the *examples* section.

The following OCL statement constraints that the values of the cells to be in the range of one to six (since it is a 6x6 puzzle).

```
context Cell
inv values : self.value > 0 and self.value <=6)
```

The following OCL excerpt illustrates the constraints that define which cells belongs to which region (notice the puzzle is split up in six regions as indicated by the bold lines in Figure 6.6.

```
context Cell
inv regions :
((self.rowIndex <= 2 and self.columnIndex <=3) implies self.region = 1)
and
((self.rowIndex <= 2 and self.columnIndex >3) implies self.region = 2)
and
((self.rowIndex > 2 and self.rowIndex <=4 and self.columnIndex <=3)
    implies self.region = 3)
and
((self.rowIndex > 2 and self.rowIndex <=4 and self.columnIndex >3)
    implies self.region = 4)
and
((self.rowIndex > 4 and self.columnIndex <=3) implies self.region = 5)
and
```

The following OCL fragment depicts specifies the existing numbers in the puzzle (for example the cell, whose row is 1 and column is 3, has a value of 6).

```
context Cell
inv numbers_exist :
((self.rowIndex = 1 and self.columnIndex = 3) implies self.value = 6)  and
((self.rowIndex = 1 and self.columnIndex = 4) implies self.value = 3) and
((self.rowIndex = 1 and self.columnIndex = 6) implies self.value = 2) and
((self.rowIndex = 2 and self.columnIndex = 1) implies self.value = 3) and
((self.rowIndex = 2 and self.columnIndex = 2) implies self.value = 2) and
((self.rowIndex = 2 and self.columnIndex = 6) implies self.value = 6) and
((self.rowIndex = 3 and self.columnIndex = 5) implies self.value = 2) and
((self.rowIndex = 3 and self.columnIndex = 6) implies self.value = 3) and
((self.rowIndex = 4 and self.columnIndex = 1) implies self.value = 2) and
((self.rowIndex = 4 and self.columnIndex = 2) implies self.value = 5) and
((self.rowIndex = 5 and self.columnIndex = 1) implies self.value = 5) and
((self.rowIndex = 5 and self.columnIndex = 5) implies self.value = 3) and
((self.rowIndex = 5 and self.columnIndex = 6) implies self.value = 1) and
((self.rowIndex = 6 and self.columnIndex = 1) implies self.value = 1) and
((self.rowIndex = 6 and self.columnIndex = 3) implies self.value = 2) and
((self.rowIndex = 6 and self.columnIndex = 4) implies self.value = 4))
```

The following OCL statement defines that all cells have different row and column indexes (if two cells have the same row and column index, then they are the same cell).

```
context Cell
inv differentCells : Cell.allInstances() -> forAll (c1, c2 : Cell |
(c1.rowIndex = c2.rowIndex and c1.columnIndex = c2.columnIndex)
implies c1 = c2 )
```

The following OCL fragment illustrates the rules of the puzzle. More specifically if two different cells belongs to the same row or column or region, they have to have a different value.

```
context Cell
inv rules : Cell.allInstances()->forAll(c1,c2:Cell | (
(c1.rowIndex = c2.rowIndex or c1.columnIndex = c2.columnIndex
      or c1.region = c2.region)
and (c1 <> c2 ))  implies c1 . value <> c2 . value)
```

B.1 Generated Alloy Model

The following listing is the Alloy code generated by the application of UML2Alloy on the UML model presented in the previous section.

```
module sudoku_puzzle
some sig Cell{
rowIndex:one Int,
columnIndex:one Int,
value:one Int,
region:one Int}

some sig Puzzle{
cells:set Cell}

fact Cell_regions_fact { all self: Cell | Cell_regions[ self ] }
fact Cell_numbers_exist_fact { all self: Cell | Cell_numbers_exist[ self ] }
fact Cell_differentCells_fact { all self: Cell | Cell_differentCells[ self ] }
fact Cell_rules_fact { all self: Cell | Cell_rules[ self ] }
fact Cell_puzzleSize_fact { all self: Cell | Cell_puzzleSize[ self ] }
fact Asso_Puzzle_puzzle_cells_Cell { Puzzle <: cells in  ( Puzzle) one->set ( Cell) }
fact AssoCustom_Puzzle_puzzle_cells_Cell { all var: Puzzle | # var . cells = 36 }

pred Cell_regions[self: Cell]{
((((((
(
int self . rowIndex =< 2) &&
(
int self . columnIndex =< 3))
 =>
 (

int self . region = 1
)


) &&
((
(
int self . rowIndex =< 2) &&
(
int self . columnIndex > 3))
 =>
 (

int self . region = 2
)


)) &&
((
((
int self . rowIndex > 2) &&
(
int self . rowIndex =< 4)) &&
(
int self . columnIndex =< 3))
 =>
 (

int self . region = 3
)


)) &&
((
((
int self . rowIndex > 2) &&
(
int self . rowIndex =< 4)) &&
(
int self . columnIndex > 3))
```

```
 =>
 (
int self . region = 4
)


)) &&
((
(
int self . rowIndex > 4) &&
(
int self . columnIndex =< 3))
 =>
 (
int self . region = 5
)


)) &&
((
(
int self . rowIndex > 4) &&
(
int self . columnIndex > 3))
 =>
 (
int self . region = 6
)


)
}


pred Cell_numbers_exist[self: Cell]{
((((((((((((((((
(
int self . rowIndex = 1) &&
(
int self . columnIndex = 3))
 =>
 (
int self . value = 6
)


) &&
((
(
int self . rowIndex = 1) &&
(
int self . columnIndex = 4))
 =>
 (
int self . value = 3
)


)) &&
((
(
int self . rowIndex = 1) &&
(
int self . columnIndex = 6))
 =>
 (
int self . value = 2
)


)) &&
((
(
int self . rowIndex = 2) &&
(
int self . columnIndex = 1))
 =>
```

```
 (

int self . value = 3
)


)) &&
((
(
int self . rowIndex = 2) &&
(
int self . columnIndex = 2))
 =>
 (

int self . value = 2
)


)) &&
((
(
int self . rowIndex = 2) &&
(
int self . columnIndex = 6))
 =>
 (

int self . value = 6
)


)) &&
((
(
int self . rowIndex = 3) &&
(
int self . columnIndex = 5))
 =>
 (

int self . value = 2
)


)) &&
((
(
int self . rowIndex = 3) &&
(
int self . columnIndex = 6))
 =>
 (

int self . value = 3
)


)) &&
((
(
int self . rowIndex = 4) &&
(
int self . columnIndex = 1))
 =>
 (

int self . value = 2
)


)) &&
((
(
int self . rowIndex = 4) &&
(
int self . columnIndex = 2))
 =>
 (

int self . value = 5
)
```

```
)) &&
((
(
int self . rowIndex = 5) &&
(
int self . columnIndex = 1))
 =>
 (

int self . value = 5
)


)) &&
((
(
int self . rowIndex = 5) &&
(
int self . columnIndex = 5))
 =>
 (

int self . value = 3
)


)) &&
((
(
int self . rowIndex = 5) &&
(
int self . columnIndex = 6))
 =>
 (

int self . value = 1
)


)) &&
((
(
int self . rowIndex = 6) &&
(
int self . columnIndex = 1))
 =>
 (

int self . value = 1
)


)) &&
((
(
int self . rowIndex = 6) &&
(
int self . columnIndex = 3))
 =>
 (

int self . value = 2
)


)) &&
((
(
int self . rowIndex = 6) &&
(
int self . columnIndex = 4))
 =>
 (

int self . value = 4
)


)
}
```

```
pred Cell_differentCells[self: Cell]{
all c1, c2: Cell | (
(c1.rowIndex = c2.rowIndex) &&
(c1.columnIndex = c2.columnIndex))
 =>
 (
c1 = c2
)



}


pred Cell_rules[self: Cell]{
all c1, c2: Cell | (
(((c1.rowIndex = c2.rowIndex) ||
(c1.columnIndex = c2.columnIndex)) ||
(c1.region = c2.region)) &&
(c1 != c2))
 =>
 (
c1.value != c2.value
)



}


pred Cell_puzzleSize[self: Cell]{
(((((((
int self . rowIndex > 0) &&
(
int self . rowIndex < 7)) &&
(
int self . columnIndex < 7)) &&
(
int self . columnIndex > 0)) &&
(
int self . value < 7)) &&
(
int self . value > 0)) &&
(
int self . region > 0)) &&
(
int self . region < 7)
}



run {} for 4 but 7 int, 36 Cell, 1 Puzzle
```

Bibliography

[1] Alloy Analyzer website. http://alloy.mit.edu/ [Accessed: April 2008]. 1

[2] UML2Alloy website. http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php. 1

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley Professional Computing. Addison-Wesley, Boston, MA, USA, 1995. ISBN 0-201-63361-2. 2.3

[4] Martin Gogolla and Mark Richters. Transformation rules for UML class diagrams. In *"UML"': '98: Selected papers from the First International Workshop on The Unified Modeling Language "UML": '98*, pages 92–106, London, UK, 1999. Springer-Verlag. ISBN 3-540-66252-9. 3.4

[5] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, London, England, 2006. 1, 2.5

[6] OMG. OCL Version 2.0, . Document id: formal/06-05-01. http://www.omg.org. 4.9, 4.11, 4.12

[7] OMG. UML Infrastructure, . Document: formal/05-07-05. http://www.omg.org. 4.15

[8] OMG. UML: Superstructure. Version 2.0, . Document id: formal/05-07-04. http://www.omg.org. 3.2, 3.4

[9] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada*, pages 94–105. IEEE Computer Society, 2003. 1

[10] Emina Torlak, Marten van Dijk, Blaise Gassend, Daniel Jackson, and Srinivas Devadas. Knowledge flow analysis for security protocols. Technical Report MIT-CSAIL-TR-2005-066, Computer Science and Artificial Intelligence Laboratory, MIT, October 2005. 6.3